Lab 3 Overview

In this lab assignment, you will do the following:

- Learn how to configure the 8051 for serial communication and how to write serial device drivers.
- Learn how to use internal XRAM and external XRAM (using the NVSRAM).
- Begin learning how to use a compiler (e.g. SDCC) to develop C programs.
- Learn how to initialize hardware properly in C.
- Continue learning about the ARM architecture and development environment.

Students must work individually and develop their own original and unique hardware/software.

The Part 1 Elements of this lab assignment are due by 4pm on **Friday, Oct. 13, 2023**.
The Part 2 Elements of this lab assignment are due by 4pm on **Friday, Oct. 20, 2023**.
The Part 3 Elements of this lab assignment are due by 4pm on **Friday, Oct. 27, 2023**.
The final submission due date (when all files must be turned in) is **11:59pm Sunday, Oct. 29, 2023**.

The cutoff date for this lab is **Saturday, November 4, 2023**.

Labs completed after the signature due date or submitted after the submission due date will be accepted, but will receive grade reductions. Labs will not be accepted after the cutoff date.

This lab is weighted as ~20% of your course grade.

Start by reading this whole lab assignment so that you understand what is expected. This lab is more time intensive than previous assignments, so you need to manage your time carefully. You should make good progress each week.

Required elements are necessary in order to proceed to the next lab assignment. Supplemental elements of the lab assignment may be completed by the student to qualify for a higher grade, but they do not have to be completed to successfully meet the minimum requirements for the lab.

ECEN 5613 students will have to complete the supplemental elements and attempt at least some of the challenges to qualify for the highest grades. To avoid any late penalties, ECEN 5613 students must obtain a TA's signature on their work by the specified signature due dates for required and supplemental elements.

If you are up for an engineering challenge, want to learn more, and have time, then attempt the optional challenge(s). You do not need to complete optional elements in order to get a signature; however, completing optional elements with good results will help your work stand out. **Students must prioritize work on required elements and supplemental elements over challenge elements.**

All items on the signoff sheet must be completed to get a signature, but partial credit is given for incomplete labs. Note that receiving a signature on the signoff sheet does not mean that your work is eligible for any particular grade; it just indicates that you have completed the work at an acceptable level.

Finish Lab #3 early, to give you more time for Lab #4 and your final project.

Lab Details

1. Review Homework assignments #6, #7, and #8, which are associated with Lab #3.
2. Refer to Lab #1 for comments regarding layout considerations, labeling, wiring, etc. All signals and pin numbers on all ICs must be labeled.
3. Learn how to configure a terminal emulator program (e.g. TeraTerm, RealTerm, PuTTY,…) on a host computer to allow you to communicate over the serial port. Be sure that you have the serial cable hooked up to the correct serial port on the computer, and that the terminal emulator is configured to use that same serial port. When first testing your hardware, you should configure the terminal emulator to communicate directly to the appropriate COM port on the PC at 9600 baud, 8 data bits, 1 stop bit, no parity and no flow control. **After you verify the hardware is working fine, you should increase the baud rate to the maximum rate that is reliable**.

   NOTE: If you do not have an RS-232 connector on your computer, you may need to use a USB to RS-232 serial adapter.

   NOTE: If you get an error message 'bytes not written correctly' when downloading your hex file with one terminal emulator program, try the other terminal emulator program and see if the error persists.

4. [**Optional**] This optional program just aids in making sure your system is wired correctly and that you are correctly initializing your processor. Write an assembly program which initializes the 8051 serial port and then continuously (in an infinite loop) transmits the character 'U'. Using an oscilloscope, verify that the transmitted patterns correspond with the ASCII value for this character and that the baud rate is correct. Verify that the characters appear on screen.

   NOTE: Make sure you understand the relation between bit rate and baud rate as well as between baud rate and the underlying carrier frequency.

   Now, modify the program to make it echo the characters it receives from the 8051 serial port. Every time a valid character is received, that same character should be transmitted back out the serial port.

NOTE: Simulation of C code is completely optional, and most students do not simulate their code once they start writing in C instead of assembly. If you want to simulate your code using Emily52, note that SFR's are **not** emulated in our version of Emily52, so you can't simulate all features of the 8051, such as the real-time aspects of serial port operation, timers, and interrupts. However, you can still use the simulator to verify much of your code. You can simulate interrupts in Emily52 by pressing 'v' for 'vector'.

5. [**Required Element**[1]] Configure your hardware to use the NVSRAM as additional XRAM data memory in your system. Prove that it responds to data reads and writes in the address range of 0x0400-0x7FFF. This will provide you with a total of 32KB of system data memory (1KB of XRAM built into the Atmel AT89C51RC2 + 31KB in the NVSRAM). [Note: Leaving 1KB of NVSRAM memory unused will simplify the decoding logic.] The lecture notes and course website have examples of how to achieve similar logic using WinCUPL's syntax.

6. [**Part 1 Required Element**[1]] Visit the PAULMON2 web site (the link is available on the course web site) and learn about this free monitor program. Download PAULMON2 (paulmon21.asm and extra.asm) and understand how to configure its assembly code to match your memory map – you will only have to make a couple of minor changes to the code. You may see that pre-assembled versions of PAULMON2 that use different memory maps are available on the PAULMON2 web site; however, don't use those versions.

   Remember that you need to configure PAULMON2 before assembling it with the AS31 assembler. As you're analyzing the equates at the beginning of the PAULMON2 code, think about using the following values for your situation - will these work for you?

```
.equ base, 0x0000 ;location for PAULMON2 (beginning of Flash)

.equ vector, 0x2000 ;location to LJMP interrupt vectors (in Flash)

.equ pgm, 0x2000  ;default location for the user program (in Flash)
                   ;pgm is where user code starts, above extra.asm

.equ bmem, 0x1000 ;beginning of memory (extra code could be in Flash)
                  ;bmem is where Paulmon starts scanning for ALL programs

.equ emem, 0x7FFF ;end of the memory (end of user code memory in Flash)
```

   Also add a new line of assembly code to the beginning of the PAULMON2 monitor program to enable the full 1KB of XRAM in the AT89C51RC2 processor. Note that when the processor comes out of reset, it has only 256 bytes of XRAM available. In order to gain access to all 1KB of internal XRAM, user code would need to program the XRS1:XRS0 bit field to 11b. (The data sheet also describes dependencies on the XRAM bit in the HSB, and the EXTRAM bit in AUXR.)

   You must assemble your own version of PAULMON2 with the AS31 assembler in order to match your memory map. The AS31 assembler and notes regarding its use are available on the course web site. Note that AS31 requires a different assembly source file syntax than the ASM51 assembler you have been using for your own code, and PAULMON2 is written with the AS31 syntax.

   Assemble your base PAULMON2 code to create one hex file. Then assemble the extra.asm code separately to create a second hex file. If using the device programmer to program your processor, you can then load both hex files into the device programmer buffer before programming your Flash. The main PAULMON2 code and the extra.asm code modules do not need to be linked together, since they are ORG'ed at different addresses and occupy different parts of the programmer buffer.

[1] Required elements are necessary in order to meet the requirements for the lab. Supplemental, optional, and challenge elements of the lab assignment may be completed by the student to qualify for a higher grade, but they do not have to be completed to successfully meet the minimum requirements for the lab.

7. [**Part 1 Required Element**[1]] Program the processor with the PAULMON2 monitor program (**including extra.asm**). If your serial port is working correctly, then when you turn on your system with PAULMON2 installed, and after you press the ENTER key, PAULMON2 should print a welcome message to your terminal emulator screen.

   Learn how to use the monitor to modify and examine hardware. Become familiar with all monitor commands, including the three extra commands ('L', 'S', 'E') provided by extra.asm. To download a hex record file to your board, use the 'D' (download) command in PAULMON2, and then use the Transfer/Send Text File menu item within the terminal emulator to transfer the hex file to your board's XRAM. Note that since the internal XRAM is located in data memory space and is not accessible with the /PSEN signal, you will not be able to execute code out of internal XRAM (i.e., you will not be able to execute the hex records that you download using the 'D' command).

   - **Determine and document the maximum baud rate at which you can reliably run PAULMON2 on your system.** Using an oscilloscope, verify that the baud rate is correct.

   Verify that you can write your entire amount of enabled XRAM (internal and external) using the PAULMON2 block fill capability.

   Step 1) Fill the entire XRAM space 0x0000 to 0x7FFF with a specific value, like 0x55.

   Step 2) Fill the rest of data memory space 0x8000 to 0xFFFF with a different value, like 0xAA. This operation shouldn't have any bad effect, as there is nothing else in your hardware that should respond to data addresses in that range (with either /RD or /WR active).

   Step 3) Dump memory from 0x0000 to 0x7FFF and verify that all XRAM cells contain 0x55. If you see blocks that do not have 0x55, something may be wrong with the hardware or with PAULMON2.

8. [**Challenge**[1]] Figure out how to use PAULMON as an on-chip debugger. You might think about where PAULMON looks for executable code when you tell it to RUN a program and try to put your code there. Demonstrate to the TA that you can both run your code normally through PAULMON and that you can single-step through the program. If you have a piece of code that is giving you problems in the future, this will be a valuable tool to help you debug the problem.

NOTE: It's possible to use part of your SRAM for code storage and be able to execute code from SRAM. In that case, you'd be able to download code to the NV-SRAM using PAULMON2, and that code would remain in memory after you turn power off to the system. When you turn power back on, the code would already be ready to run. Slight adjustments to your memory map and chip select and output enable logic might be needed. A portion of your SRAM would need to be enabled by /PSEN and /RD to respond during both code read and data read operations.

NOTE: For the rest of the semester, use the Atmel FLIP utility to program your code into the 8051 processor flash memory.

NOTE: For the following C programs, remember that your C code has to finish initializing the AT89C51RC2 chip. In order to gain access to all 1KB of XRAM, user code would need to program the XRS1:XRS0 bit field to 11b. (The data sheet also describes dependencies on the XRAM bit in the HSB, and the EXTRAM bit in AUXR.) Be sure to review the sdccman.pdf file regarding the *_sdcc_external_startup()* function. If you specify an XRAM size of 0x400 on the linker command line, you need to make sure there are actually 0x400 bytes of XRAM enabled in the hardware. Likewise, if you specify an XRAM size of 0x8000 on the linker command line, 0x8000 bytes of XRAM must be enabled.

9. [**Part 1 Required Element**[1]] Learn how to use SDCC to generate hex records for your hardware using the large (or small) memory model. Compile some simple code (use the SDCC Syntax Examples file for reference) and know how to verify that SDCC has been configured with the correct addresses for ROM/RAM by analyzing the SDCC output files. Verify that the correct addresses were generated in your code listing file (.rst) and hex record file (.ihx). You should also examine the other output files to see how SDCC has allocated space for objects in memory (check the other output files, such as .mem, .lnk, and .map). Make sure you understand how the memory models differ and what the pros and cons are for using one relative to another.

   NOTE: You are allowed to use SDCC with any IDE, makefile, or command line system you like to develop your C code for the semester. For your professional development as an embedded systems engineer, you are highly encouraged to understand how to build code both by using an IDE and by using a makefile.

   If you use an IDE, learn how to do the following:

   (a) Open a new SDCC project and configure your compiler/linker options.
   (b) **Use a subfolder for your source files (e.g. Sources or src).**
   (c) Edit your C code and examine the files in your project.
   (d) Compile your C code and create a hex file. Examine the .rst listing file.

   NOTE: There is a command line FLIP utility called "batchisp" which is included in the standard FLIP installation. Students can create makefiles that can support automatic compilation and flashing to the chip and can also add make options to automatically open a serial terminal.

   Example programming command:

   batchisp -device at89c51rc2 -hardware RS232 -port COM3 -baudrate 57600 -operation erase f memory flash blankcheck loadbuffer ./$(BIN_DIR)/$(BIN).hex program verify start reset 1

   Example serial terminal command:

   ttermpro c=3 baud=57600

- **Submit your files electronically.** Submit your Paulmon files (your modified .asm and resulting hex files) and your experimental test SDCC files (.c, .h, .rst, .mem, .map, .ihx) to Canvas in the Lab 3 Part 1 folder.

COMPLETE AND SUBMIT ALL THE ITEMS ABOVE FOR THE PART 1 ELEMENTS CHECKPOINT. TA'S WILL SIGN OFF YOUR WORK DURING THE PART 2 ELEMENTS SIGNOFF.

NOTES:

Various IDE's, like Code::Blocks, are free.

All Code::Blocks project settings, including compiler and linker settings, can be edited from the Project>Properties menu item (this is also available on right-click on the project in the Projects list pane).

You must backup your files for this course regularly using storage other than our ECEE 1B28 lab computers. For source code control, the free Git or Subversion (SVN) version control systems are recommended: **https://git-scm.com/** and **http://subversion.tigris.org** and **http://tortoisesvn.tigris.org/**

If you want to use a stand-alone version of make, you are encouraged to download GNU make from the course web site, since it is free. Makefile examples are available on the course web site.

If you want to use a non-SDCC 8051 compiler, such as Keil, this is acceptable with some caveats:

1) You need to have the compiler available in the lab during signoffs, so that you can build code and demonstrate knowledge of the tool set. You will need to have a demo version or licensed version available on your notebook computer, if the tools are not installed on the computers in the ECEE Dept.
2) You will not get as much help from the TAs or instructor if they are not familiar with your tool set.
3) You must realize that examples given in class or on the web will be specific to SDCC. We cannot support more tools than those related to SDCC.

NOTE: Notes about SDCC and SDCC syntax examples are available on the course web site. If you're using SDCC, you should absolutely read these notes, as they should save you a significant amount of time. Refer to these! Make any necessary updates to the examples in order to conform to the ESE C-Coding Style Guideline.

TIP: To speed up testing, you can create text files with specific contents and send the text files from the PC to your embedded system to simulate a large number of characters being entered on the keyboard**. A fast baud rate will save you time.** Note that TeraTerm allows you to drag and drop a file into the terminal emulator window.

TIP: When loading hex files with PAULMON2, use the fastest baud rate to shorten download time.

NOTE: Whenever you're not using a monitor and you're burning your code into Flash or an EPROM, remember to initialize the serial port prior to using any serial input/output function calls (e.g. write your own serial port initialization routine).

NOTE: Flow control only tells the transmitter to stop transmitting - it doesn't tell that system to stop receiving. You can send flow control characters from your terminal emulator by pressing Control-S (^S) and Control-Q (^Q). XOFF = DC3 = ^S = 0x13 XON = DC1 = ^Q = 0x11

NOTE: A formatted version of the AS31 assembler documentation is also available on the course web site. AS31 (not ASM51) is used to assemble the PAULMON2 source code.

NOTE: When using the PAULMON2 monitor, which has automatic baud rate detection, you must cycle power and press ENTER after each baud rate change: just pressing the 8051 run time reset button will not clear the current baud rate.

NOTE: You can demonstrate your hardware functionality using PAULMON2. Make sure you are very familiar with all commands for the PAULMON2 monitor. To demonstrate your XRAM and RS-232 hardware, modify values in your XRAM using the monitor, and then verify those values by reading them. Consider developing just one or two programs which allow you to demonstrate all optional hardware and firmware functionality. This will reduce the amount of time it takes to get signed off.

NOTE: When asking the TA's and instructor for help with your program via e-mail, be sure to attach all relevant files to your e-mail. Attach the commented source files (.c, .h), and the .rst, .mem, and .map files. You may also need to attach a PDF of your schematic.

NOTE: The following C program has a little complexity. Don't try to code it all at once. Instead, identify elements of the assignment, and then write and debug in an incremental manner. For example, here are some suggestions on how one might approach the assignment:

1) Become comfortable with the compiler and your build method (e.g. IDE or makefiles). Write very simple programs (like an 8051 pin toggling program with no circuits connected to the GPIO pin) to gain confidence and understanding of the tools. Refer to the SDCC syntax examples that are available on the course web site.
2) Learn how to initialize the hardware properly, including the serial port. Put initialization code that must execute early, such as XRAM enablement, in the _sdcc_external_startup() function. Put other initialization code in your own init() function that you call early in your main() function.
3) Learn how to use putchar() or putch() to output single characters to the terminal emulator. Make sure that your init() function properly initializes the TI flag.
4) Learn how to input single characters from the terminal emulator, using the getchar or getch() functions. Use the putchar() or putch() function to echo received characters to the screen.
5) Learn how to create a buffer using the malloc() command. Learn how to free up memory space once the buffer is no longer needed. Learn how to use pointers to write to and read from a buffer.
6) Write a function that can print out buffer contents, using the specified format.
7) Identify other independent elements of the program assignment, and write functions to implement those elements.
8) After you have the individual functions working, then concentrate on how to use those functions to complete the programming assignment.
9) Utilize version control and save your code to a remote server with clear commit comments whenever your code reaches a milestone.

USE AN INCREMENTAL APPROACH TO SAVE YOURSELF MUCH DEBUGGING TIME!

START YOUR ASSIGNMENT EARLY, AS IT MAY TAKE MORE TIME THAN YOU MAY THINK.

OPTIONAL: To make signoffs efficient and to use less of the limited signoff time for code re-flashing with FLIP, students should look for ways to minimize the number of times the processor flash memory must be updated with different code images during the signoff process.

It is perfectly acceptable for students to show Paulmon and C code separately. However, as a show of extra effort, students could load their flash memory with a single code image containing Paulmon and the remaining code exercises, and then run the programs stored in flash memory through Paulmon.

Running your C program through Paulmon should obviate the need for using FLIP to load multiple individual programs during the signoff. Unlike downloading and executing code from the XRAM through Paulmon, running code from the flash memory through Paulmon can be done by following these steps:

1) Change the linker options so that your code starts from an address above 0x2000 (e.g. your linker options might look like this --code-loc 0x3000 --code-size 0x5000…)
2) Enable the processor's internal XRAM in Paulmon. Also, do not initialize the serial port in your C program; just let Paulmon take care of the serial communications.
3) Use FLIP to load your C code but make sure the "Erase" checkbox is unchecked as we don't want to erase Paulmon and extra from the flash memory (assuming Paulmon is already installed).
4) Your code can now be run through Paulmon by using the 'J' command to jump to the location you specified in the linker options.
5) You may also use a Paulmon header so that you can run your program using the 'R' command. Since the Paulmon header is written in assembly, you need to find a way to add it to your C code.

10. [**Part 2 Required Element**[1]] Write a C program which first allocates a heap of size 5000 bytes, and then prompts the user to specify a buffer size between 32 and 4800 bytes, where the buffer size must be evenly divisible by 16. The program must then allocate a buffer (buffer_0) of the requested buffer size in XRAM using the `malloc()` function. The program must then malloc in XRAM a second buffer (buffer_1) which is also equal to the requested buffer size. If the malloc fails for any buffer, then any successfully allocated space must be freed and the user must be prompted to choose a smaller buffer size. Use pointers to external memory to access the buffers.

NOTE: Older versions of SDCC use different methods for creating the heap. Make sure you follow the correct procedure for your version of SDCC.

The program must then prompt the user to enter characters. For this lab, 'storage' characters include upper case letters. All other characters, including punctuation, single space, lower case letters, numbers, and special control characters, such as line feed, carriage return, horizontal tab, etc., are considered to be 'command' characters. Every time any character is received at the 8051 serial port, that same character must be transmitted back out the 8051 serial port. Also, if the character is a storage character, it must be stored in buffer_0. Command characters are not stored. The use of buffers other than buffer_0 is not specified in this assignment, and those other buffers may be used for any other purpose.

If the '+' character is received, the program must prompt the user to specify a buffer size between 20 and 400 bytes inclusive. The program must then try to allocate a new buffer (buffer_n) of the requested buffer size in XRAM using the `malloc()` function. If the malloc fails for the buffer, the user must be informed. The user should be able to create new buffers as long as there is space on the heap. The program must immediately report to the screen a message indicating the success or failure regarding the new buffer creation. If the '-' character is received, the program must prompt the user to specify a valid buffer number. If the buffer number is valid, the program must then delete that buffer and use the `free()` function to free up its space from the heap. Valid buffer numbers for this particular case include any current buffers except 0 (i.e. buffer_0 may not be deleted).

The program must keep count of how many total characters and how many storage characters are received. Every time the character '?' is received, the program must provide a report on the heap, including information about each buffer currently in the heap, including: buffer #, buffer start address, buffer end address, total allocated size of the buffer (in bytes), number of storage characters currently in that buffer, and number of free spaces remaining in that buffer. Note that the heap report will change as storage characters are stored in and emptied from buffer_0. The program must also report the total number of characters received since the last '?'. The C program must clearly report all these numbers on the terminal emulator screen, along with descriptive text. The program then must empty the buffer by transmitting all the characters which were stored in buffer_0, with a maximum of 32 ASCII characters displayed on each line of the screen. buffer_0 will then be empty. The quality of your user interface will be a factor in your grade.

If buffer_0 fills completely before a '?' command is received, any excess character subsequently received for that buffer is echoed out the serial port, but is not added to that buffer (it is discarded). Once the '?' command is received, then buffer_0 is emptied, as described above.

If the '=' character is ever received, the program must display the current contents of buffer_0 in hex, **but must not empty the buffer** – the data will remain in the buffer until the buffer emptied in response to a '?' command. Display the data on the PC screen in hexadecimal (not ASCII), with 16 bytes of data per line, in the following format (AAAA is address, DD is data):

AAAA: DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD

If the '@' character is ever received, the program must immediately use the `free()` function to free the heap space being utilized by all current buffers that have been allocated. The program shall then start over from the beginning and ask the user to specify a new buffer size.

Hint: You could create and send text files to your embedded system using the terminal emulator in order to speed testing, verifying that your code properly handles large blocks of characters. You could create text files specific to test cases that you want to verify with your system; this allows you to do regression testing each time you make changes to your code and you want to check a previous test case.

Note: In order for you to maximize your grade, your code must gracefully handle erroneous inputs, such as the user trying to specify a buffer size outside the allowed range. You should verify that your code handles cases like this correctly before you get signed off. During signoff, you will be evaluated on the quality of your code at that point in time. You do not get multiple chances to get feedback from the TA or instructor and make improvements to your code based on that feedback prior to your signoff, so be well prepared before your signoff. Be able to explain how SDCC allocated heap space for your program. Code with a good user interface, robust error handling, and good structure and comments will score higher than code lacking these features. Please try to follow the ESE C-Coding Style Guideline that was discussed in class and posted on Canvas.

Virtual Debug Port

NOTE: Integrate this virtual debug port into your buffer code above.

You must also implement a 'virtual' debug port using a DEBUGPORT macro, to be used with a logic analyzer. For instance, write a value to data memory at some unused fixed address, perhaps 0xFFFF. That address is in address space reserved for peripherals and is not used by most students.

Note: Remember that you have set up 1KB of internal XRAM in the AT89C51RC2 MCU, so if you don't choose to use address 0xFFFF make sure to choose a data memory address equal to or above 0x0400. There will be no activity on the address lines if you choose a data memory address between 0x0000 and 0x03FF which will be the memory space of internal XRAM.

Implement a debug macro something like this pseudocode:

```
#ifdef  DEBUG
#define DEBUGPORT(x)  dataout(0xFFFF,x);   // generates a  MOVX 0FFFFh,x where x is an 8-bit
value #else
#define DEBUGPORT(x)    // empty statement, nothing passed on from the preprocessor to the compiler
#endif
```

In order to use your debug port, put DEBUGPORT() statements with unique data values in each function you want to trace, similar to the pseudocode below.

```
main()
{
DEBUGPORT(0x00)
myfunction1();            //various code and function calls here
DEBUGPORT(0x01)
myfunction2();            //various code and function calls here
DEBUGPORT(0x02)
}

void  myISR(void)  __interrupt  (1)
{
DEBUGPORT(0xF0)
//ISR code here
DEBUGPORT(0xF1)
}
```

Trigger the logic analyzer on write cycles to data memory address 0xFFFF. There is no hardware at address 0xFFFF, and that address will be used exclusively for logic analyzer debug support.

To enable debugging, you will need to #define DEBUG. The DEBUGPORT function can be easily removed from your code if you don't define DEBUG or if you #undef DEBUG. You can do that if you have timing sensitive code that should not be influenced by the debug statement execution time.

This virtual debug port is just as useful as the '374 debug port from Lab #2, and has the benefit of not requiring any extra silicon on your board, thus saving cost, space, and power. This virtual debug port has a performance advantage over debugging using the serial port, since execution of the MOVX instruction will be much faster than the polling involved in sending characters to the serial port.

Idea: You can also optionally choose to write to more than one data address if you think it would be valuable, by either using a macro definition like DEBUGPORT((y),(x)) that takes a variable address or by implementing a second debug macro that uses a different fixed address. Perhaps one address could be used for ISR's and one address could be used for non-interrupt code.

- Submit your neat and fully commented program electronically. Make regular backups of your code.

COMPLETE ALL THE ITEMS ABOVE FOR THE PART 2 ELEMENTS SIGNOFF SUBMISSION.

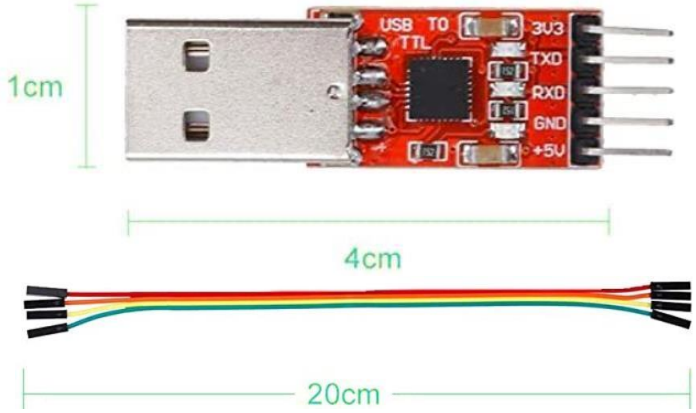
COMPLETE ALL THE ITEMS BELOW FOR THE PART 3 ELEMENTS SIGNOFF SUBMISSION.

11. **[Part 3 Required Element[1]]:** Continue learning about your ARM dev board.

IF YOU HAVE THE STM32 BOARD:

- Obtain a USB to TTL converter from the TA and use that converter to interface the STM32 UART pins with your system.
    - Ensure the USB to TTL converter is displayed as a COM port in the Device Manager when you connect with the system
    - Connect USB to TTL converter Tx to Rx of STM32's UART and vice-versa using the jumper wires. **It is recommended using STM32 UART2 which uses the PA2 and PA3 pins on the dev board.** Ensure you have provided a common ground.

      One converter is: https://www.amazon.com/IZOKEE-CP2102-Converter-AdapterDownloader/dp/B07D6LLX19



- Perform an echo function using UART communication. Upon receiving characters, transmit those characters back out the UART, "echoing" what the user types. Well-designed programs should implement interrupt-driven TX and RX of characters. Reasonable buffering of TX and RX should be used to ensure no characters are lost from frequent transmissions.
- Generate a PWM signal with a default 60% duty cycle using one of the STM32 GPIO pins. The PWM signal should be shown on the oscilloscope during the signoffs. Take note of the logic high voltage you see being driven on the STM32 GPIO pin. Compare this to what you've seen as a logic high voltage from the GPIO pins on the 8051. Configure the push-button (labeled as B1 in the dev board internally routed to GPIO PA0) to increase the output duty cycle by 10% on each press until it reaches 100%, then the duty cycle should decrease by 10% on each press. Consequently, when the duty cycle reaches 0%, it should start increasing by 10% on each press. Report these changes over the UART. When the program receives a character 'P', print the current duty cycle (i.e. 70%) out to the UART. Format output from the program in a way that is easy to understand. Think about how you interact with a command-line interface and how other programs format their outputs to distinguish themselves from user input. Consider putting outputs from the program on new lines, for example.
- Write 2 commands, one that increases the PWM duty cycle by 5% and the other that decreases the duty cycle by 5%. For example, let the duty cycle increase by 5% when the program receives the 'A' command via UART and decrease by 5% when the program receives the 'B' command.

STM32 References:

1) STM32 folder inside the Files folder on our course Canvas site
2) User Manual:
   https://www.st.com/resource/en/user_manual/dm00148985-discovery-kit-with-stm32f411ve-mcustmicroelectronics.pdf
3) Reference Manual:
   https://www.st.com/resource/en/reference_manual/dm00119316-stm32f411xc-e-advanced-arm-based-32bit-mcus-stmicroelectronics.pdf
4) Description of HAL and low layer Drivers https://www.st.com/resource/en/user_manual/dm00105879-description-of-stm32f4-hal-and-ll-driversstmicroelectronics.pdf
5) STM32 HAL Examples:
   https://github.com/STMicroelectronics/STM32CubeF4/tree/master/Projects/STM32F411EDiscovery/Examples/UART
6) STM32 bare metal Implementation
   https://github.com/a5221985/tutorials/blob/master/Embedded_Systems/embedded_systems_bare_metal_programming_ground_up.md

Note that students are highly encouraged to do bare metal programming in this course and not to rely on development tools that abstract the lower levels of the hardware and firmware. In this course, your work will be evaluated as having more engineering content if you are developing all the code rather than relying on a tool to develop the code for you.

12. **[Part 3 Supplemental Element[1]]:** Demonstrate additional ARM functionality on your board.

    Extra functionality can be added to ARM code above to demonstrate additional learning and effort on your part. Features should be easy to demonstrate during sign-off and able to be verified by the TA during grading. Ideas include (but are by no means limited to):

    STM32:

    Use the PWM output and route it to one of the GPIO pins either [PD12 / PD13 / PD14 / PD15]. As the PWM increases/decreases on every push button depending on the current duty cycle, the LED brightness also should change in accordance with the current duty cycle. Commands can be developed to alter the period of the PWM signal.

13. **[Part 3 Supplemental Element[1]]:**

Demonstrate other features of the Atmel AT89C51RC2 as described below.

Each Programmable Counter Array (PCA) compare/capture module can be programmed in any one of the following modes:

- Rising and/or falling edge capture
- Software timer
- High-speed output
- Pulse width modulator (PWM)
- Module 4 can also be programmed as a watchdog timer.

In this element, you need to prove that you can use at least three of the PCA modes above. The minimum requirement is one PCA module configured in PWM mode and Module 4 configured as a watchdog timer.

Note: For this element, configure the system clock in **X2 mode**, not standard mode.

To demonstrate the PWM mode, configure one pin as a pulse width modulation output with a 33% duty cycle and allow the PWM Counter to continue to run while the system is in Idle mode. Use $F_{CLK\ PERIPH}/2$ as the clock source for the PCA timer.

Prove that your hardware works by providing firmware which demonstrates the required functionality. Your program must provide a well-designed user menu which allows the user to perform the following options:

- Run PWM (turn on PWM output)
- Stop PWM (turn off PWM output)
- Set $F_{CLK\ PERIPH}$ at the minimum frequency supported by the CKRL register
- Set $F_{CLK\ PERIPH}$ at the maximum frequency supported by the CKRL register
- Enter Idle mode (set IDLE bit in PCON register)
- Enter Power Down mode (set PDE bit in PCON register)

Prove that you have successfully entered Idle mode by examining ALE, PSEN*, and XTAL2 with a digital logic probe or oscilloscope. Prove that the PWM Counter continues to function while in Idle mode. Prove that you can exit Idle mode and continue running code when an interrupt is received at an external interrupt while the processor is in Idle mode. You can use C code or inline assembly to generate the ISR to handle the event.

Prove that you have successfully entered Power Down mode by examining ALE, PSEN*, and XTAL2 with a digital logic probe or oscilloscope.

Use the logic analyzer or oscilloscope to verify what happens to both the PWM output and the ALE frequency when you switch the CKRL register between max frequency and min frequency. Explain your findings.

Be able to describe in detail how the chosen PCA modes work.

- Submit an electronic copy of your neat and fully commented programs.

Optional Challenges

If you are up for an engineering challenge, want to learn more, and have time, then attempt one or more of the optional challenges. You do not need to complete these optional elements in order to get a signature; however, completing optional elements with good results will help your work stand out. **Students must prioritize work on required elements and supplemental elements over challenge elements**. Note that some students develop their own challenges to help their work stand out.

Students should expect to receive less TA and instructor guidance on challenges than on the required or supplemental elements.

14. [**Challenge**[1]] Write code that runs on your 8051 to demonstrate how to utilize the built in ISP functions via calls to the Atmel AT89C51RC2 bootloader (see the API calls in Table 74 in the Atmel AT89C51RC2 data sheet). You may also want to refer to other documents on the Atmel web site regarding the Bootloader and In System Programming. CAUTION: The API calls will allow you to overwrite flash memory and change hardware registers that are otherwise only accessible via special tools like the Phyton device programmer. By using the API calls, it is possible to put your processor into a bad state that may require you to try to recover it using the Phyton device programmer.

15. [**Challenge**[1]] In an assembly file, write a function in assembly language that takes three integer parameters (e.g. param1, param2, and param3), calculates the value of ((param3 * param2) mod param1) and returns the integer result. Prove that by using assembly language, you can access the parameters in your assembly language function. Prove that you can call the assembly language function from a C file and that the returned value from the assembly function is correct. You must have two files – one written in C and one written in assembly. You will need to figure out how to compile, assemble, and link them correctly. During your signoff, explain what the stack frame pointer is and how it is used. Prove that you can call and pass a parameter to a C function in the .c file from the assembly language function in the .asm file. Prove that you can access a global variable in the C file from the assembly language function. *This challenge element may be completed on either the 8051 or ARM system.*

16. [**Challenge**[1]] Using C, implement an interrupt-driven serial port driver (for both transmit and receive) instead of using polled I/O. Implement a FIFO and prove improved system performance (higher bandwidth or lower overhead) as compared with a polled I/O approach. Explain the engineering differences between a polled and interrupt-driven implementation. *This challenge element may be completed on either the 8051 or ARM system.*

17. [**Challenge**[1]] By studying the SDCC library source code for the 8051 and writing your own test code, learn the details of how SDCC does heap management, specifically for the `malloc()` and `free()` functionality. Given the program described in Required Element 10, evaluate how the heap is managed for the following sequence:

    Step 1) The heap is created with 5600 bytes.
    Step 2) Buffers 0 and 1 are created with 2400 bytes each.
    Step 3) Buffer 2 of 200 bytes is created using the '+' command.
    Step 4) Buffer 3 of 300 bytes is created using the '+' command.
    Step 5) Buffer 2 is deleted using the '-' command.

    What does the heap free memory pool look like after the sequence above?

    Describe what would happen for the three following possible cases:

    a. If Step 6 of the sequence was to create a buffer of 100 bytes, where would it be allocated?
    b. If Step 6 of the sequence was to create a buffer of 210 bytes, where would it be allocated?
    c. If Step 6 of the sequence was to create a buffer of 800 bytes, where would it be allocated?

    Describe the problem of memory fragmentation when using dynamic memory allocation in C.

Submission Questions

18. [**Part 3 Required Element**[1]] As part of your submission, provide answers to the following questions:

    a) What operating system (including revision) did you use for your code development?
    b) What compiler (including revision) did you use?
    c) What exactly (include name/revision if appropriate) did you use to build your code (what IDE, make/makefile, or command line)?
    d) Did you install and use any other software tools to complete your lab assignment?
    e) Did you experience any problems with any of the software tools? If so, describe the problems.

NOTE: Make copies of your code, SPLD code, and schematic files and save them as an archive. You will need to submit the Lab #3 files electronically at the end of the semester.

Submission Instructions

Instructions: Print your name and sign the honor code pledge. Separate the signoff sheet from the rest of the lab and turn in a scan of the signed form, the items in the checklist below, and the answers to any applicable lab questions in order to receive credit for your work. No cover sheet please. **Submit all items electronically via Canvas (https://canvas.colorado.edu).** Please follow the instructions given below:

1. Create a folder called "lastname_lab_3" where "lastname" is your last name. You will create subfolders inside this unzipped folder, ending with a folder structure like the following:
   - lastname_lab_3
     - schematic
     - spld_code
     - 8051_code
     - stm32_code
     - lab_writeup
2. Please include a legible and easy-to-view copy of your complete and accurate schematic (all old/new components shown) in pdf format inside the sub-folder named "schematic".
3. Include your spld source file (i.e. .pld file) and any spld simulation file (i.e. .si file) in the sub-folder named "spld_code".

✈ Submit all of your 8051 code files in the sub-folder named "8051_code". Include full copy of fully, neatly, clearly commented source code (including C and header files, and .RST, .MEM, and .MAP files). Ensure your code is neatly formatted and easy to read, and that each and every source file has header comments that identify the author and any leveraged code the file contains. Code should be well organized in folders. If you submit a PDF of your code to improve its appearance, be aware that you must also submit the original source code files as well. Your lab grade will not be released unless you have submitted your original source files.

4. Submit all of your ARM code files in the sub-folder named "stm32_code". Ensure that these files are neatly formatted and easy to read.
5. Submit your write-up in the sub-folder named "lab_writeup" either in Word or PDF format. Your write-up should follow these guidelines:
   - Include scan of signed and dated signoff sheet as the first page. Include the backside of the signoff sheet as well if there are comments written by the TAs. Do not add any cover sheet.
   - Write-up should be concise and less than 5 pages (excluding the signoff sheet). Use only font size 11 or 12 to maintain legibility. Do not describe the objectives of the lab or any redundant information; focus on your results and answers. Submit the SDCC command line options you used when building your code. If you used a makefile, submit a copy of that makefile.
   - Include any important analyses and timing diagrams
   - Include all the calculations that are required to be done for answering the questions that have been asked in the sign-off sheets or lab assignment.
   - Include screen-shots (if any) in PDF format in this folder itself.
   - Comment on any significant learnings from the lab assignment.
   - Include the answers to any submission questions in the lab assignment.
   - Must include clear high-resolution pictures of top and bottom sides of the 8051 board you assembled for this lab, including wiring and labels; wire wrapping and soldering must be clearly visible when you zoom in – **make sure your pictures are in focus**. Use JPEG format.
6. Submit a zipped version of the folder as "lastname_lab_3.zip" as an attachment via the Canvas interface. Please use only the ".zip" file format when submitting files.
7. Please contact the TA's or instructor in case you have any doubts regarding submissions.

NOTE: Make and save archive copies of your code, SPLD code, and schematic files. You need to submit the Lab #3 files electronically, now and at the end of the semester.

ECEN 5613 **Lab #3 Signoff Sheet** **Fall 2023**

You will need to obtain the signature of your instructor or TA on the following items in order to receive credit for your lab assignment. Print your name below, sign the honor code pledge, circle your course number, and then demonstrate your working hardware & firmware in order to obtain the necessary signatures.

Student Name: _____

Honor Code Pledge: "On my honor, as a University of Colorado student, I have neither given nor received unauthorized assistance on this work. I have clearly acknowledged work that is not my own."

Student Signature: _____

Signoff Checklist

Part 1 Elements

- ❑ Schematic of acceptable quality (all components shown)
- ❑ Pins and signals labeled, decoupling capacitors, and two 28-pin wire wrap sockets present on board
- ❑ Very good knowledge of a terminal emulator
- ❑ Demonstrates all 32KB of XRAM in memory map are functional, including monitor block fill command
- ❑ Using PAULMON2, demonstrates highest baud rate as: _____
- ❑ Knows how to use SDCC [IDE or make optional]

_____
TA signature and date

Part 2 Elements

- ❑ Knows how to analyze output files (.RST, .MEM, .MAP) for correct addresses
- ❑ C serial program and virtual debug port functional and code commented
- ❑ Hex display of buffer contents

_____
TA signature and date

Part 3 Required and Supplemental Elements

- ❑ Required ARM code integration and execution
- ❑ 8051 PWM control works correctly, X2 mode
- ❑ Correctly enters Idle mode and exits via external interrupt 1
- ❑ Correctly enters Power Down mode
- ❑ All other PCA software menu items function correctly
- ❑ Good understanding of PCA modes
- ❑ Good user interface; program is easy to use

Instructor/TA Comments: □ □ □

_____
TA signature and date

FOR INSTRUCTOR USE ONLY Part 1 and 2 Elements	Not Applicable	Below Expectation	Meets Requirements	Exceeds Requirements	Outstanding
Schematics, SPLD code	☐	☐	☐	☐	☐
Hardware physical implementation	☐	☐	☐	☐	☐
Part 1 Required Elements functionality	☐	☐	☐	☐	☐
Sign-off done without excessive retries		☐	☐		
Student understanding and skills	☐	☐	☐	☐	☐
Overall Demo Quality (Part 2 elements)	☐	☐	☐	☐	☐

FOR INSTRUCTOR USE ONLY Part 3 Elements	Not Applicable	Below Expectation	Meets Requirements	Exceeds Requirements	Outstanding
Part 3 Required Elements functionality	☐	☐	☐	☐	☐
Supplemental Elements functionality	☐	☐	☐	☐	☐
Student understanding and skills	☐	☐	☐	☐	☐
Overall Demo Quality (Part 3 elements)	☐	☐	☐	☐	☐

Comments:

- ❑ Optional Challenge: PAULMON2 RUN command
- ❑ Optional Challenge: ISP API calls
- ❑ Optional Challenge: C and Assembly interfacing
- ❑ Optional Challenge: Serial ISR
- ❑ Optional Challenge: SDCC heap memory management analysis